同济大学

TONGJI UNIVERSITY

编译原理课程设计

课题名称	编译原理课程设计
副 标 题	类 C 编译器设计与实现
学　　院	同济大学
专　　业	计算机科学与技术
学生姓名	郑星云
学　　号	2151610
指导教师	卫志华
日　　期	2024 年 3 月 9 日

目 录

目 录

1 需求分析

1.1 程序功能概述

本课程设计旨在使用高级程序语言开发一个一次性完成的简易 C 语言编译器。该编译器具备以下功能：词法分析、语法分析、符号表管理、中间代码生成和目标代码生成，以及函数调用。

用户需要提供一个包含代码的文本文件。编译器将对代码进行语法和语义错误分析，若代码无误，将其编译成中间代码（四元式格式）及目标代码（MIPS 汇编语言）；若存在错误，则返回错误信息。

1.2 支持的语法和特性

- 支持整型变量的声明、定义、赋值以及基本运算，并允许在过程中嵌套定义具有相同名称的变量。
- 布尔表达式使用短路优化策略。
- 支持循环（for/while）和条件（if）控制语句。
- 实现函数的声明、定义和调用，并支持函数的递归调用。
- 整个编译过程只需对源程序和各个阶段中间结果扫描一次，确保编译效率。
- 支持单行和多行注释。
- 报错提示，编译错误时会提供相关错误信息供用户校对。
- 内置 `read` 和 `write` 函数，允许汇编执行过程中与用户直接交互。

详细测试数据见报告的调试分析部分。

1.3 输入形式

本编译器接受一个文本文件作为输入，该文件应包含用简易 C 语言编写的源代码。输入文件的具体要求如下：

- **编码格式**：文本文件应采用 UTF-8 编码格式，以保证各种字符和符号的准确读取。
- **语言特性**：输入的源代码需遵循本编译器支持的语法和特性，包括变量的声明与赋值、基本运算、控制语句（如 if、for、while）和函数的声明、定义与调用等。
- **变量范围**：支持的变量类型仅限于整型。整型变量用于存储整数值，范围依据使用的编程语言和编译器实现细节而定，但通常符合 32 位整数的标准范围（$-2,147,483,648$ 至 $2,147,483,647$）。
- **函数特性**：函数支持递归调用和嵌套定义，但必须确保递归深度和嵌套层次不超过编译器处理能力，以避免栈溢出等问题。
- **内置函数**：源代码中可以使用内置的 `read` 和 `write` 函数与用户进行交互。

输入文件应严格遵守上述规范和限制，以确保编译器能够正确地处理和编译源代码。不符合规范的代码可能会导致编译错误或运行时异常。

1.4 输出形式

- **中间代码**：编译过程将源代码转换成四元式格式的中间代码，便于后续生成目标代码。中间代码将详细反映原程序的逻辑结构和操作细节。
- **目标代码**：最终输出为 MIPS 汇编语言格式的目标代码，可在支持 MIPS 架构的模拟器或硬件上执行。
- **错误信息**：如果源代码存在语法或语义错误，编译器将输出具体的错误信息，帮助用户定位并修正问题。

2 概要设计

2.1 任务分解

本编译器设计分为以下核心任务：

1. **词法分析**：
   通过扫描源代码，将其分解为一系列记号（Token），为接下来的语法分析阶段铺垫。
2. **语法分析**：
   本编译器采用 LR1 分析方法对源代码进行语法解析。利用扩展巴克斯范式（EBNF）编写的文法文件，构建 ACTION 和 GOTO 表，作为语法分析的基础。为提高编译效率，文法文件初次解析后将缓存结果，以便后续使用时直接读取，避免重复解析。
   在语法分析过程中，依次处理词法分析阶段识别的记号。分析器将维护状态栈和符号栈两个栈结构，并根据当前输入符号及栈顶状态查询 ACTION 和 GOTO 表，执行相应的移进或归约操作。
3. **中间代码生成**：
   利用语法制导的方法生成四元式格式的中间代码，为目标代码生成做准备。
   在语法分析的归约操作后立即执行对应的语义动作，这一步骤涉及传递符号属性及生成四元式。
4. **目标代码生成**：
   目标代码的生成以函数为单位。在一个函数的实现被完全归约后，触发目标代码生成器，将该函数的中间代码转换为 MIPS 汇编语言代码。根据四元式中的操作类型采用相应的策略进行代码生成。
   这一阶段需要处理临时寄存器的分配、栈空间的管理、函数调用时参数的传递及现场的保存与恢复等问题。
5. **符号表管理**：
   在整个编译过程中，管理所有变量和函数的声明及作用域，确保其在合适的上下文中被引用。这一管理机制对于生成准确和高效的目标代码至关重要。
6. **错误处理**：
   在进行语法和语义分析的过程中，编译器会及时识别并报告错误，提供错误信息以帮助用户定位并修正代码问题。

2.2 主程序流程

以下是根据 2.1 对主程序流程的详细说明：

1. **启动**：编译过程的初始化阶段，构造必要的实例对象。

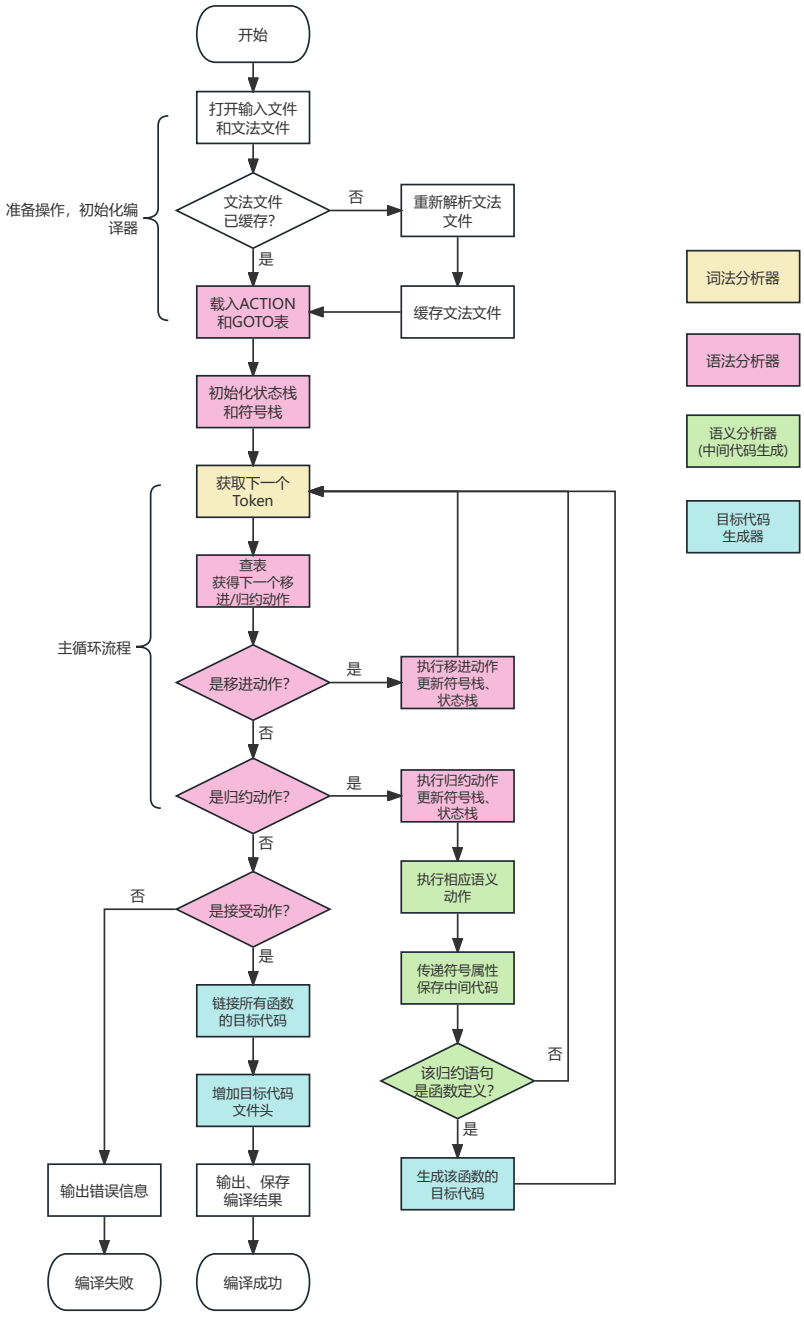

图 2.1 主程序流程图

2. **读取源代码文件和文法文件**：主程序首先读取用户提供的源代码文件和用于语法分析的文法文件。
3. **检查文法文件是否已解析**：
    - 如果是首次编译或文法文件有更改，编译器将解析文法文件，构建 ACTION 和 GOTO 表，并将解析结果缓存。
    - 如果已有解析缓存，则直接读取缓存内容，跳过解析过程。
4. **获取下一个 Token**：编译器将从源代码中获取下一个 Token，这是编译过程中的一个基本步骤，涉及将源代码分解成一系列的标记。
5. **查看动作表是否为移入动作**：
    - 如果动作表指示进行*移入动作*，编译器将 Token 移入符号栈，返回第四步。
6. **查看动作表是否为归约动作**：
    - 如果动作表指示进行*归约动作*，编译器将按照某个产生式规则归约已识别的 Token 序列，并执行语义动作。
    - **执行语义动作**：传递语法的符号的继承属性给父节点。如果归约的语句是函数定义语句，生成该函数的目标代码；否则返回第四步。
7. **是否接受动作？**：确定是否接受这个动作。
    - 如果是接受动作，链接所有函数的目标代码，追加目标代码的文件头，编译成功。
    - 如果不是接受动作，意味着编译错误，无法被语法分析器解析。则输出对应的错误信息，编译失败。

2.3 模块间调用关系

如 2.2 所示，该项目构建了以下主要模块，以及它们之间的调用关系：

- **编译器（Compiler）**：作为项目的核心模块，由主函数直接调用，负责整个编译流程的协调与管理。
- **词法分析器（Lexer）**：实现词法分析功能，是编译流程的初始阶段。它负责解析源代码文本，识别并生成一系列记号（Token），供后续分析使用。
- **语法分析器（LR1Parser）**：负责语法分析，构成编译过程的核心环节。在遇到需要进行语义分析的归约动作时，会调用语义分析器。
- **语义分析器（SemanticAnalyzer）**：当语法分析器执行归约动作时触发，主要处理变量定义、类型检查等语义规则。在处理函数定义等特定语义动作时，将调用目标代码生成器以产生相应的机器代码或中间代码。
- **目标代码生成器（InstructionGenerator）**：负责最终的目标代码生成。它根据语义分析的结果，将中间代码转换为目标平台上的汇编代码。该过程主要由语义分析器触发，并将生成的

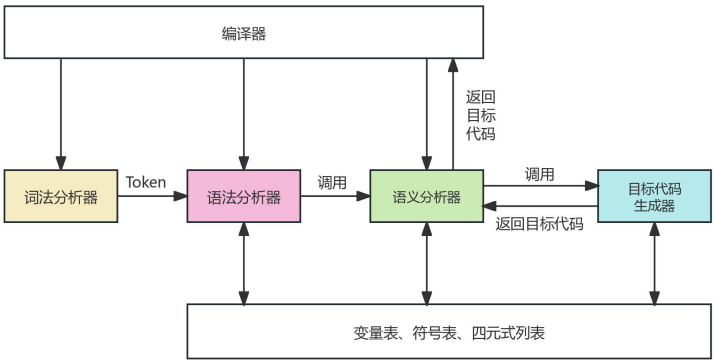

图 2.2 模块调用关系

目标代码经由语义分析器反馈给编译器模块。

2.4 数据类型的定义

在编译器实现过程中，将一些关键概念抽象为特定的数据类型，以下详细解释这些类型及其属性：

2.4.1 记号（Token）

代表词法分析阶段解析得到的每个独立单词的标识。

表 2.1 记号（Token）的属性

属性名	类型	描述
type	TokenType	Token 的类别，包括字面量、关键字、操作符、界符、EOF 等
value	std::string	Token 的实际文本值

2.4.2 符号（Symbol）

在编译过程中非常重要的一种数据类型，用于语法分析阶段作为分析基础，并在语义分析阶段用于传递和保存属性文法信息。

表 2.2 符号（Symbol）的属性

属性名	类型	描述
type	SymbolType	符号的类别，如终结符、非终结符或空串
literal	std::string	符号的字面值，例如非终结符 ”<S>” 或终结符 ”T_INT”
real_value	std::string	符号的实际值，例如变量名 ”a” 或整数值 ”3”
meta	MetaData	符号的附加属性，为自定义类型

注：其他数据类型的表格可以按照上述模式进行构建。

通用数据容器（MetaData）

`MetaData` 类扮演着属性传递的中介角色。它是一个灵活的键值对容器，使用 `std::unordered_map<std::string, std::any>` 来存储数据，允许以字符串作为键，任意类型作为值。这种设计既保证了数据传递的灵活性，也支持了类型安全的数据检索。

此类型的设计避免了需要为每一种符号类型定义一个类，简化了程序结构并提高了扩展性。

A.

2.4.3 产生式（Production）

产生式表示文法的各个规则。

表 2.3 产生式（Production）的属性

属性名	类型	描述
lhs	Symbol	产生式的左侧，为一个符号（非终结符）
rhs	std::vector<Symbol>	产生式的右侧，为符号序列（包含非终结符和终结符的有序集合）

2.4.4 动作（Action）

语法分析阶段 ACTION 表的条目，包括移进、归约、接受、错误等类型。

表 2.4 动作（Action）的属性

属性名	类型	描述
type	ActionType	动作类型
number	size_t	仅移进类型有效，表示将要移进的新状态编号
production	Production	仅归约类型有效，表示用于归约的产生式

2.4.5 LR1 项目（LR1Item）

LR1 分析方法中的项目，包括一个产生式、一个点（表示当前分析进度）和一个向前看符号。

表 2.5 LR1 项目（LR1Item）的属性

属性名	类型	描述
production	Production	产生式
dot_position	size_t	点的位置，决定了 LR1 项目的状态
lookahead	Symbol	向前看符号

2.4.6 变量（Variable）

语义分析阶段中，用户定义的变量的抽象表示。

此处解释一些属性的作用：

- deep：因为该编译器允许变量的嵌套定义，如：
  子过程允许定义和父过程同名变量，在调用时自底向上检索变量表中的同名变量，实现“就近原则”的变量嵌套定义。

表 2.6 变量（Variable）的属性

属性名	类型	描述
id	std::string	变量的唯一标识符
name	std::string	变量名
type	VariableType	变量类型，仅支持 int 和 void
deep	size_t	变量的嵌套深度，0 表示全局变量
param	int	标记变量是否为函数参数

```
int main() {
  int a = 1;
  {
    int a = 2;
    print(a); // 此时输出 2
  }
  print(a); // 此时输出 1
}
```

- param：标记变量是否为函数形参，-1 表示不是形参，否则表示函数的第几个形参
  用于目标代码生成时的函数调用参数传递，param 等于 03 可以直接传入$a 寄存器，大于 3 则要压入栈中保存。

2.4.7 函数（Function）

语义分析阶段中，用户定义的函数的抽象表示。

表 2.7 函数（Function）的属性

属性名	类型	备注
return_type	VariableType	函数的返回类型
name	std::string	函数名
formal_variables	std::vector<Variable>	函数的形参列表

2.4.8 四元式（Quater）

四元式形式的中间代码抽象类。

表 2.8 四元式（Quater）的属性

属性名	类型	备注
op	std::string	四元式操作符
arg1	std::string	四元式第一个参数（可为空）
arg2	std::string	四元式第二个参数（可为空）
result	std::string	四元式目标参数（可为空）
meta	MetaData	四元式的额外属性

2.4.9 Avalue 和 Rvalue

这两个数据类型用于寄存器分配过程。简单来说，Avalue 某个变量此刻的状态，Rvalue 表示某个寄存器此刻的状态。

表 2.9 Avalue 的属性

属性名	类型	描述
variable_id	std::string	变量 ID，引用 Variable
variable_name	std::string	变量名
load	bool	标记变量是否加载在寄存器中
offset	size_t	变量在栈中的偏移（若未加载在寄存器中）
pos	size_t	变量在哪个寄存器中（若加载在寄存器中）
start_point	size_t	变量生命周期开始时间
end_point	size_t	变量生命周期结束时间
param	int	标记变量是否为函数参数

表 2.10 Rvalue 的属性

属性名	类型	描述
variable_id	std::string	寄存器中变量 ID，引用 Variable
start_point	size_t	寄存器中变量生命周期开始时间
end_point	size_t	寄存器中变量生命周期结束时间
busy	bool	标记寄存器是否被占用

装 订 线

3 详细设计

3.1 词法分析

3.1.1 接口设计

词法分析器类提供两个主要的公共接口：

1. `getNextToken()`：返回当前位置起的下一个 Token 记录。
2. `getCurrentColRow()`：当发生错误时，返回当前扫描到的行号和列号，并打印最后两行的代码内容以便调试。

3.1.2 核心逻辑

词法分析器主要由两个关键成员变量组成：一个字符串，存储了用户输入的整个代码文件；一个指针，指示当前扫描到的字符位置。

此外，词法分析器维护了几个变量来记录当前的行号、列号以及当前行和上一行的文本内容，这些信息用于编译错误时提供参考。

`getNextToken()` 函数的执行逻辑大致如下：

1. 跳过任何空白字符，包括空格、换行等。
2. 跳过注释内容。
3. 检查是否到达文件末尾。如果是，则结束程序；如果不是，继续执行。
4. 根据当前字符预判下一个 Token 的可能类别：
    - 如果当前字符是数字，则下一个 Token 很可能是数字类型，此时应用处理数字的策略。
    - 如果当前字符是字母或下划线，下一个 Token 可能是标识符或关键字。首先检查是否为已知的关键字，如果是，则返回关键字 Token；否则，返回标识符 Token。
    - 对于其他类型的字符，如运算符和界定符，按照相应的逻辑处理。

3.1.3 支持的符号

词法分析支持的符号包括：

- 操作符（Operators）：
    - 算术运算符：`+`, `-`, `*`, `/`, `%`（求余）
    - 赋值运算符：`=`
    - 比较运算符：`==`, `!=`, `<`, `<=`, `>`, `>=`
    - 逻辑运算符：`&&`（逻辑与），`||`（逻辑或），`!`（逻辑非）

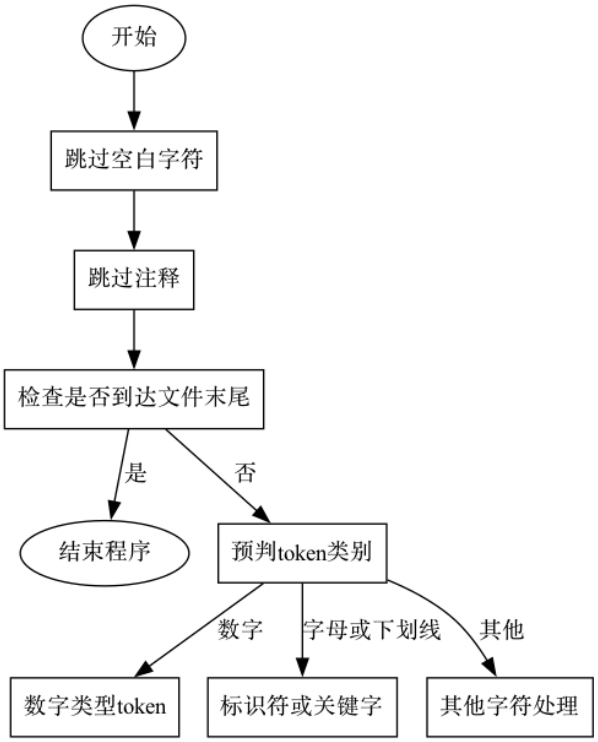

图 3.1 `getNextToken` 函数流程图

- 关键字（Keywords）：
  - 控制流程语句：`if`, `else`, `while`, `for`, `return`
  - 数据类型：`int`, `void`
- 界定符（Delimiters）：逗号、分号、大括号、小括号、等标准程序结构界定符：`,`, `;`, `{`, `}`, `(`, `)`
- 字面量和标识符：数字（整数）、标识符（如变量和函数名）
- 注释：单行注释: `//`、多行注释：`/**/`
- 空白字符

3.2 语法分析

采用的是 LR1 分析方法，该方法是一种自底向上的语法分析技术，用于构造解析特定文法的有限自动机。以下是实现 LR1 语法分析的详细设计步骤：

3.2.1 初始化及构造 FIRST 集

1. **初始化 FIRST 集**：遍历所有文法产生式，为每个符号初始化其 FIRST 集。对于终结符和空串，直接将其加入到自身的 FIRST 集中。对于非终结符，则初始化其 FIRST 集为一个空集。
2. **迭代计算 FIRST 集**：
   1. 对每个非终结符，遍历所有以该非终结符为左侧（lhs）的产生式。

2. 将产生式右侧首个符号的 FIRST 集合并到左侧非终结符的 FIRST 集中。
3. 如果 FIRST 集发生了变化，则继续迭代；如果没有变化，说明 FIRST 集计算完成。

3.2.2 构造 LR1 分析表

计算完 FIRST 集后，进入构造 LR1 分析表的阶段，该阶段的关键在于构建识别特定前缀的有限自动机的状态转移。

1. **构建初始状态**：以文法的起始产生式构建初始状态的闭包，作为自动机的起始状态。
2. **状态转移处理**：
   - 移进（*SHIFT*）：对于每个终结符，构建新的状态集，计算闭包，检查是否已存在。不存在则加入状态集，同时更新 ACTION 表添加移进动作。
   - 待约（*GOTO*）：对每个非终结符，同样构建新的状态集并计算闭包，如不存在则加入，并在 GOTO 表中添加转移动作。
   - 接受（*ACCEPT*）：在 ACTION 表中对应项标记为接受动作。
   - 归约（*REDUCE*）：对归约项目，在 ACTION 表中添加归约动作，指明使用哪个产生式进行归约。

3.2.3 进行语法分析

构建完 ACTION 和 GOTO 表后，可以开始语法分析过程。该过程通过三个栈来维持：状态栈、符号栈和输入栈，初始化时状态栈包含初始状态 0，符号栈为空。

按照以下规则进行分析：

- 移进（*SHIFT*）：
  - 读取输入符号，根据 ACTION 表执行移进操作，更新状态栈和符号栈。
  - 输入符号出栈。
- 归约（*REDUCE*）：
  - 根据归约产生式，从符号栈和状态栈中弹出对应数量的元素。
  - 将归约产生式的左侧符号压入符号栈。
  - 查找 GOTO 表获取新状态，压入状态栈。
- 接受（*ACCEPT*）：
  - 如果按照 ACTION 表的指示执行接受动作，则分析成功完成。
- 错误处理：
  - 如果在 ACTION 表找不到当前状态和符号对应的动作，分析失败，输出错误信息。

通过以上步骤，LR1 分析方法能够准确地进行语法分析，为后续的语义分析和代码生成提供了基础。

3.3 语义分析

在编译的一遍扫描方法中，语义分析采用属性文法和语法制导翻译策略，紧密结合语法分析进行。每当语法分析过程中发生归约时，就执行与该归约产生式相关的语义动作。这些动作负责计算和传递符号的属性值，同时维护变量和函数的符号表，并生成四元式形式的中间代码。

该阶段核心难点在于如何为各种语法特性设计相应的制导语法。下面分语法特性来分析。

3.3.1 变量嵌套声明

A. 文法及归约

文法结构如下：

```
<var_declaration> ::= <type_specifier> T_IDENTIFIER T_SEMICOLON
<type_specifier>  ::= <simple_type>
<simple_type>     ::= T_VOID | T_INT
```

通过终结符确定变量的基本类型（如 `int` 或 `void`），并将这个类型信息作为属性传递给 `<type_specifier>`。

在归约时，接收类型说明符和标识符（变量名），从 `<type_specifier>` 接收类型属性，并将其与变量名一起，添加到变量表中。

B. 变量嵌套深度（deep）

变量的嵌套深度由代码块的花括号 `{}` 控制。每当遇到一个左花括号 `{` 时，嵌套深度加一；每当遇到一个右花括号 `}` 时，嵌套深度减一。

变量表中记录了每个变量的嵌套深度。在添加新变量时，会检查同一嵌套深度内是否存在同名变量，如果存在，则报错；否则，加入变量表。

当嵌套深度减小时，需要从变量表中移除所有超出当前深度的变量，表示这些变量的作用域已结束。

```c
{
    int a = 1;
    {
        int a = 3;  // 合法，因为是在新的嵌套深度中
        {
            int a = 5;  // 合法，进一步嵌套
            print(a);  // 输出 5，访问的是最内层的 a
        }
        {
            int a = 7;
            print(a); // 输出 7
        }
        print(a);  // 输出 3，访问的是中间层的 a
    }
    print(a);  // 输出 1，访问的是最外层的 a
    int a = 3;  // 报错，因为在相同的嵌套深度内已存在名为 a 的变量
}
```

代码 3.2 示例代码：变量嵌套声明和作用域

C. 示例分析

代码 3.2，变量表变化过程如图 3.3

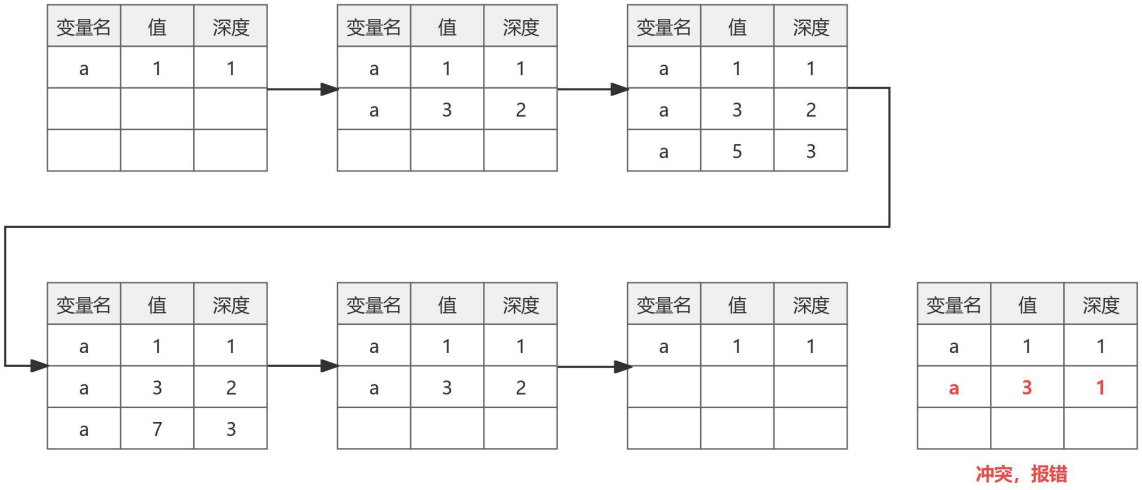

图 3.3 变量表变化过程

这个例子展示了语义分析如何处理变量的作用域和嵌套声明，通过deep 属性来确保变量声明的正确性，并处理作用域结束时的变量生命周期。这种机制有效地支持了 C 语言中的变量作用域规则，保证了语义分析的准确性和高效性。

3.3.2 普通运算语句

普通运算语句在语义分析阶段的核心任务是处理表达式的计算逻辑，并生成相应的中间代码，即四元式。下面详细介绍这一过程的实现细节。

A. 文法及归约

普通运算表达式的文法设计，通过层级结构实现运算符优先级，定义如下：

```
<expression> ::= <additive_expression> | <var> T_ASSIGN <expression> | ...
<additive_expression> ::= <term> | <additive_expression> <addop> <term>
<term> ::= <factor> | <term> <mulop> <factor>
<factor> ::= T_LEFT_PAREN <expression> T_RIGHT_PAREN | <var> | T_INTEGER_LITERAL | ...
<addop> ::= T_PLUS | T_MINUS
<mulop> ::= T_MULTIPLY | T_DIVIDE | T_MOD
<var> ::= T_IDENTIFIER
```

归约过程根据文法逐层处理，从 `factor` 获取最基本的计算单位（如变量、整数等），`term` 和 `additive_expression` 通过 `mulop` 和 `addop` 实现了乘除与加减操作的归约。

B. 中间代码生成

在归约过程中，针对运算表达式生成中间代码，即四元式。四元式格式为 `(Oper, Arg1, Arg2, Result)`，其中 `Oper` 是操作符，`Arg1` 和 `Arg2` 是操作数，`Result` 是运算结果。

对于二元运算，如 `a + b * c`，中间代码生成步骤如下：

1. 首先处理乘法运算 `b * c`，生成四元式 `(*, b, c, #1)`，其中 `#1` 是临时变量存储 `b * c` 的结果。
2. 然后处理加法运算 `a + #1`，生成四元式 `(+, a, #1, #2)`，其中 `#2` 是临时变量存储最终结果。

对于赋值运算，例如 `a = b + c` 的四元式表示：

1. 首先计算二元加法运算，生成四元式 `(+, b, c, #1)`，其中 `#1` 存储 `b + c` 的结果，
2. 然后然后赋值给 `a`，生成四元式 `(=, #1, , a)`。

C. 临时变量管理

中间代码生成过程中，对于临时变量的管理是一个关键点。每次运算产生的临时结果都存储在一个新的临时变量中，临时变量以 `#` 开头，后跟递增的编号，以区分不同的运算中间结果。

对于每个新的运算，生成一个新的临时变量作为运算结果的存储位置。

3.3.3 布尔表达式

布尔表达式的中间代码生成关键在于实现短路逻辑，并优化生成的代码以提高执行效率。以下详细介绍布尔表达式中间代码的生成方法。

A. 文法及归约

布尔表达式的文法设计如下，以支持基本的逻辑运算及短路逻辑：

```
<bool_expression> ::=
        T_LEFT_PAREN <bool_expression> T_RIGHT_PAREN |
        <additive_expression> <relop> <additive_expression> |
        <bool_expression> T_AND <bool_expression> |
        <bool_expression> T_OR <bool_expression> |
        T_NOT <bool_expression>
```

该文法允许基本的逻辑运算，并通过组合使用这些逻辑运算符，实现复杂的布尔表达式。

B. 中间代码生成

对于基本的比较运算，生成判断指令和跳转指令，分别对应两个出口，例如 `a < b` 可生成如下中间代码：

```
Oper   Arg1 Arg2 Result
j<     a    b    true_exit
j                false_exit
```

其中，`true_exit` 是满足条件时跳转的标签，`false\_exit` 是不满足条件时跳转的标签

每一个布尔表达式都维护所有真假出口的四元式编号，当使用逻辑运算符连接的时候，管理连接这些出口。

- **逻辑 AND**

  利用短路特性，第一个表达式为假则整个表达式为假，不需要计算第二个表达式。例如 `exp1 AND exp2`，如果 `exp1` 为假，则直接跳转到假出口，真出口连接到 `exp2` 的判断。

- **逻辑 OR**

  类似于 AND，但逻辑相反。如果 `exp1` 为真，则整个表达式为真，直接跳转到真出口，假出口连接到 `exp2` 的判断。

- **逻辑 NOT**

  逻辑非运算，将真假出口互换。

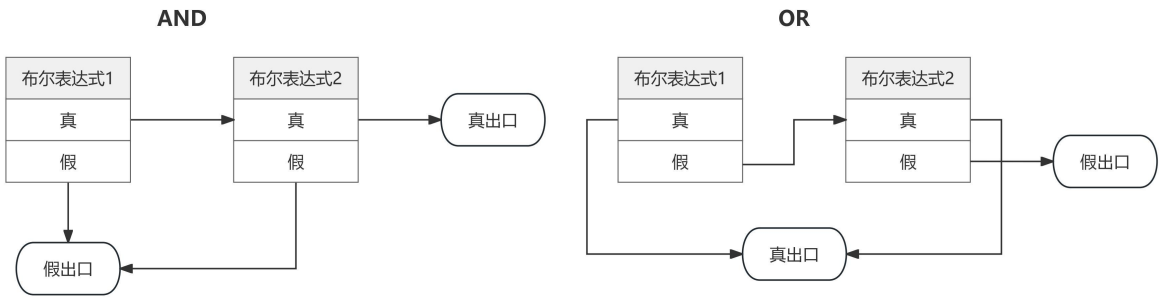

图 3.4 布尔表达式真假出口示意图

3.3.4 控制语句

控制语句分为条件控制语句和迭代控制语句两种，迭代控制语句又分为 for 循环和 while 循环两种。

每个控制语句就是以一种固定的方法管理其条件（布尔表达式）的真假出口指向哪里，再在特定位置增加无条件跳转语句。

- **条件控制语句（if-else）**

  条件为真时执行 if 语句体，条件为假时根据 else 的存在选择跳转。例如：

```
if condition goto L1
[else 部分代码]
goto L2
L1: [if 部分代码]
L2: [后续代码]
```

- **for 语句**

  for 循环的控制逻辑分解为：初始化、条件检测、循环体执行和循环末尾的迭代表达式。例如：

```
[初始化代码]
L1: if condition goto L2
goto L3
L2: [循环体代码]
[迭代表达式代码]
goto L1
L3: [后续代码]
```

- **while 语句**

  while 循环直接依据条件表达式进行循环体的控制。例如：

```
L1: if condition goto L2
goto L3
L2: [循环体代码]
goto L1
L3: [后续代码]
```

装 订 线

3.3.5 函数

在编译过程中，函数的处理是编译器工作的重要组成部分。

语义分析器中有一个函数表来维护所有函数。在声明时加入函数表，调用时在函数表中查询相应的函数。

函数处理包括两个主要方面：函数的声明（定义）和函数的调用。这两个方面在语义分析阶段都需要生成相应的中间代码，以便于后续的目标代码生成。

A. 声明和定义

函数的声明和定义通过两套独立的文法处理，允许在函数尚未定义前进行声明，支持函数间的相互调用。

函数声明主要涉及三个方面：返回类型、函数名、参数列表。其中，参数列表可能为空，参数间以逗号分隔，记录了各参数的类型及变量名。目前只支持整型（INT）和空类型（VOID）作为返回类型。

对于函数定义，它是由一系列语句组成的代码块，这些语句被转化为四元式形式，构成函数的实现体。例如：

```
int add(int a, int b) {
    return a + b;
}
```

此函数定义生成的四元式包括：

Oper	Arg1	Arg2	Result
=	param0		a
=	param1		b
+	a	b	#1
return	#1		

其中，形参的获取通过特殊的四元式表示，如 `param0` 和 `param1` 被赋值给 `a` 和 `b`。函数必须以 `return` 语句结束，其四元式直接表示返回操作，与函数声明的返回类型相匹配。

B. 调用

函数调用时需传递实际参数。例如：

```
int main() {
    int a = 1; int b = 2;
    add(a, b);
    return 0;
}
```

生成的四元式如下：

Oper	Arg1	Arg2	Result
=	1		a
=	2		b
param	a	2	0
param	b	2	1
call	add	2	#1
return	0		

在调用函数之前，会生成特定的 param 四元式为每个实参设置位置，后跟一个 call 四元式执行实际的函数调用。

这里的 param 四元式指定了传入参数的值、参数总数和参数位置，call 四元式指明了被调用的函数名、参数总数及函数返回值的存储位置。

通过这种方式，函数的声明、定义及调用通过中间代码精确表示，为目标代码生成提供了必要的信息，同时也支持了高效的函数调用和参数传递机制。

这里额外添加的四元式是为了方便目标代码生成器采取不同的生成策略，详细内容在目标代码生成章节介绍。

3.4 目标代码生成

目标代码生成阶段的主要任务是将四元式形式的中间代码转换为可执行的目标代码。这一过程涉及到根据操作符（op）选择相应的转换策略，处理不同类型的操作数（arg1, arg2），以及管理变量的存储和寄存器的分配。

四元式 (op, arg1, arg2, result) 根据操作符 op 的不同，采用不同的目标代码生成策略。操作数 arg1 和 arg2 可能是局部变量、临时变量、立即数，或者为空（在某些操作，如赋值操作中，只需要单个操作数）。result 通常是局部变量或临时变量。

变量在目标代码中的访问分为“读取”和“写入”两种情况：

- **读取**：当变量作为操作数出现在 arg1 或 arg2 位置时。
- **写入**：当变量作为结果出现在 result 位置时。

变量应在被读取之前被写入，即必须首先初始化。变量的“生命周期”从它第一次被写入直到最后一次被访问。

为了访问变量，系统必须先将其加载到寄存器中。如果变量还没有分配寄存器，系统将为其分配一个新的寄存器；如果已经分配了寄存器，则使用该寄存器。由于寄存器数量有限，当活跃变量数量超过寄存器数量时，某些不活跃的变量需要被临时保存到内存（即栈）中。当这些变量再次被访问时，它们将从内存中读回到寄存器。

3.4.1 栈空间管理

在函数执行期间，栈空间的管理是目标代码生成过程中的一个关键环节，负责局部变量的存储、函数调用时参数的传递以及保存寄存器状态等。以下是对栈空间管理策略的概述：

A. **初始化**

函数执行开始时，需要对栈帧（Stack Frame）进行初始化，设置基指针（Frame Pointer, $fp）和栈指针（Stack Pointer, $sp）：

```
1        move    $fp, $sp
```

这里，$fp 记录了当前栈帧的底部，作为局部变量和传入参数的访问基点，类似于 x86 架构中的 bp 寄存器。$sp 指向栈的顶部，随着局部变量的分配和释放动态变化。

B. **保存（压栈）**

在进行函数调用或需要保留寄存器值时，将寄存器的值保存到栈上：

```
1        sw      $t0, 0($sp)
2        addi    $sp, $sp, -4
3
4        sw      $t1, 0($sp)
5        addi    $sp, $sp, -4
```

使用 `sw` 指令将寄存器 `$t0` 的值存储到栈顶位置，随后通过 `addi` 指令调整栈指针 `$sp`，为新的数据腾出空间。这个过程通常称为“压栈”。

经过上述代码后，栈空间如下表所示（假设地址从 100 开始）

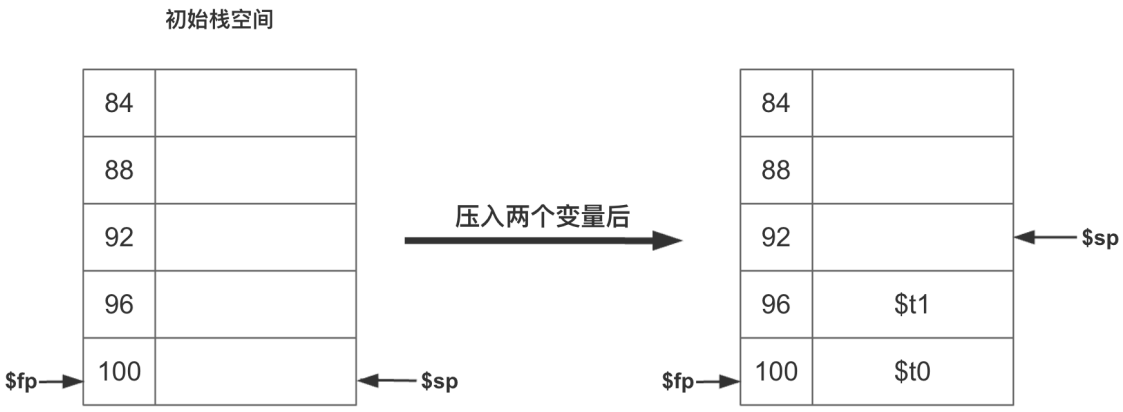

图 3.5 Enter Caption

C. **访问**

局部变量和保存的寄存器值可以通过栈基指针 `$fp` 加上一个偏移量来访问，偏移量表示该变量或值相对于栈帧底部的位置：

```
1        lw   $t0, 0($fp)    # 访问栈帧底部的第一个局部变量或保存的值
2        lw   $t1, -4($fp)   # 访问栈帧底部向上 4 字节处的局部变量或保存的值
```

使用 `lw` 指令通过 `$fp` 寄存器和偏移量间接寻址，实现对栈上特定位置数据的读取。

通过上述管理策略，编译器能够有效地在函数执行过程中管理栈空间，确保局部变量的正确存储和访问，以及函数调用期间寄存器状态的保存和恢复。这些操作确保了程序的正确执行和数据的安全，是目标代码生成中不可或缺的一部分。

3.4.2 临时寄存器分配策略

在目标代码生成阶段，有效地分配和管理临时寄存器对于提高程序执行效率至关重要。MIPS架构提供了一组 `$t0` 至 `$t9` 的临时寄存器用于这一目的。下面是一个简化的寄存器分配策略，旨在最大化利用有限的寄存器资源，同时处理变量的溢出情况。

A. **管理表维护**

Rvalue 表：记录每个临时寄存器的占用情况，包括它当前是否被使用，以及被哪个变量占用。

Avalue 表：记录每个变量的存储位置，可能是寄存器或内存。如果变量存储在寄存器中，记录其寄存器编号；如果溢出到栈中，记录其在栈中的偏移量。

B. **寄存器分配流程（基于线性扫描算法）**

1. **初始化活跃集合：**算法开始时，没有任何寄存器被分配，所以活跃集合（存储当前被使用的寄存器信息）是空的。

2. **遍历生存期：**对于每个变量的生存期（即程序中变量从被定义到最后一次使用的区间），按照它们开始时间的顺序进行排序和处理。

3. **移除过期变量：**在尝试为新的变量分配寄存器之前，检查活跃集合中是否有变量的生存期已经结束。如果有，就把它们对应的寄存器释放回寄存器池，这样就可以重新被分配了。

4. **寄存器分配决策：**

- 如果所有寄存器都已经被占用（即活跃集合的长度等于寄存器总数），那么就需要选择一个变量进行溢出处理，将它存储到内存中，而不是寄存器。
- 如果还有空闲寄存器，就从寄存器池中移除一个寄存器，分配给当前变量，并更新活跃集合，保持按结束时间排序。

5. **处理寄存器溢出：**如果没有空闲寄存器，需要选择一个变量进行溢出。这里选择的是活跃集合中结束时间最晚的那个变量（因为它占用寄存器的时间最长）。如果这个变量的结束时间比当前变量的结束时间晚，就把寄存器分配给当前变量，并把那个长期变量溢出到内存。否则，当前变量被溢出到内存。

算法 1 线性扫描寄存器分配

```
1: procedure LinearScanRegisterAllocation
2:     active ← {}
3:     for i in live intervals in order of increasing start point do
4:         ExpireOldIntervals(i)
5:         if length(active) == R then
6:             SpillAtInterval(i)
7:         else
8:             register[i] ← a register removed from pool of free registers
```

```
9:              add i to active, sorted by increasing end point
10:         end if
11:     end for
12: end procedure
13: procedure ExpireOldIntervals(i)
14:     for j in active, in order of increasing end point do
15:         if endpoint[j] ≤ startpoint[i] then
16:             return
17:         end if
18:         remove j from active
19:         add register[j] to pool of free registers
20:     end for
21: end procedure
22: procedure SpillAtInterval(i)
23:     spill ← last interval in active
24:     if endpoint[spill] > endpoint[i] then
25:         register[i] ← register[spill]
26:         location[spill] ← new stack location
27:         remove spill from active
28:         add i to active, sorted by increasing end point
29:     else
30:         location[i] ← new stack location
31:     end if
32: end procedure
```

现在对上述分配寄存器的过程进行简化：

1. 如果这个变量已经被分配，直接返回对应的变量信息。
2. 清理超过生命周期的变量。
3. 如果没有空闲寄存器了，根据线性扫描算法选择结束时间最晚的变量溢出。如果结束时间最晚的有多个，选择其中开始时间最早的。
4. 遍历所有寄存器，将空闲寄存器分配给这个变量。

当一个变量需要溢出时，其值被保存到栈空间中。编译器会统计溢出的所有变量数量，在函数一开始就预分配相应大小的栈空间。一个变量在首次溢出到栈时会被分配一个固定的栈位置，后续的溢出会使用相同的位置，避免重复分配。

这种寄存器分配方法不仅确保了有效的寄存器利用，还通过合理的溢出处理机制保持了程序的高效运行。采用线性扫描算法为每个变量动态选择最优的存储位置，有效平衡了寄存器资源的使用和内存访问的开销。通过这种策略，即便在寄存器数量有限的情况下，也能保证程序的运行效率。

3.4.3 赋值运算与二元表达式

下面介绍具体四元式如何转换成目标代码。

A. **赋值运算**

立即数赋值：当赋值操作的源是立即数时，可以直接使用 MIPS 的 `li` 指令（load immediate）进行赋值。

例如，四元式 (=, 5, , a) 可以转换为以下汇编指令：

```
1        li $t0, 5       # 将立即数 5 加载到寄存器 t0 中
```

变量赋值：对于变量到变量的赋值操作，需要先确保源变量被加载到某个寄存器中，然后通过 `move` 指令将值赋给目标变量所在的寄存器。

在执行操作前，需为变量分配寄存器。若变量已有分配的寄存器，直接使用；若无，则根据寄存器分配策略分配新寄存器。

B. **二元运算表达式**

对于加、减、乘、除等二元运算，处理过程需保证两个操作数同时在寄存器中：

1. **寄存器保留**：在分配寄存器之前，确保至少有两个寄存器是空闲的，以便同时容纳两个操作数。
2. **参数分配**：首先为两个操作数分配寄存器。若操作数是变量或临时变量，则加载到寄存器中；若操作数是立即数，则可直接在后续运算指令中使用。
3. **目标分配**：为运算结果分配寄存器。考虑到目标寄存器可以是其中一个操作数的寄存器，因此在分配目标寄存器时应优先重用操作数寄存器，以减少不必要的数据移动。

示例：对于四元式 (+, a, b, c)，转换为 MIPS 目标代码的步骤如下：

```
1        lw $t0, 0($fp)  # 加载变量 a 到寄存器 t0
2        lw $t1, -4($fp)  # 加载变量 b 到寄存器 t1
3        add $t2, $t0, $t1  # 将 t0 和 t1 中的值相加，结果存储在 t2
```

在这个过程中，若寄存器资源紧张，可能需要执行溢出操作，将部分不活跃的变量或之前的运算结果暂存到内存中，留出空间进行当前运算。通过上述策略，编译器能够高效地将四元式转换为目标代码，同时优化寄存器的使用和减少内存访问次数。

3.4.4 转移语句

转移语句在控制流程中起着至关重要的作用，包括条件转移和无条件转移两种类型。在将四元式转换为目标代码时，需要特别处理这些转移指令，确保程序能够按照预期的逻辑进行跳转。

A. 预处理

在生成目标代码之前，先对所有四元式进行一遍扫描，为所有的转移目标位置增加标签（label）。这些标签在汇编代码中作为跳转目标，确保无条件跳转和条件跳转指令能够正确地将控制权转移给程序的其他部分。

B. 无条件转移

无条件转移语句相对简单，直接转换为汇编语言中的跳转指令。例如，四元式形式的无条件跳转（`j，_，_，label`）直接对应于 MIPS 汇编指令 `j label`，其中 `label` 是事先处理过程中加入的对应标签。

C. 条件转移

条件转移语句需要根据具体的条件选择相应的汇编跳转指令。MIPS 架构提供了一系列条件分支指令，如 `beq`（等于时跳转）、`bne`（不等于时跳转）、`blt`（小于时跳转）等，用于处理不同的条件判断。

例如，条件转移四元式（`j<，a，b，label`）表示如果 `a < b` 则跳转到 `label`。在 MIPS 汇编中，这可以转换为 `blt $a，$b，label`，其中 `$a` 和 `$b` 分别是存储变量 `a` 和 `b` 值的寄存器。

D. 示例

考虑以下简单的控制流程示例，使用条件和无条件跳转来控制程序执行流程：

```
if (a < b)
    goto label1;
goto label2;
label1:
// 执行某些操作
label2:
// 继续执行
```

对应的四元式和转换后的 MIPS 汇编代码可能如下：

四元式：

```
(j<, a, b, label1)
(j, _, _, label2)
```

MIPS 汇编：

```
    blt $a, $b, label1
    j label2
    label1:
    # 执行某些操作
    label2:
    # 继续执行
```

通过这种方式，编译器将高级语言中的控制流程转换为汇编语言中的跳转指令，实现了程序逻辑的正确控制。正确处理条件转移和无条件转移语句是生成有效目标代码的关键步骤之一。

3.4.5 函数

函数调用在汇编语言中需要特别注意现场保存与恢复、参数传递以及函数调用和返回的正确执行。以下是在 MIPS 体系结构下函数调用的具体处理方法。

A. 现场保存与恢复

在函数调用前，需要保存调用者的环境，确保函数执行完毕后能够恢复到原始状态。

- **保存操作**：
  1. 保存临时寄存器：将所有使用中的临时寄存器的内容保存到栈上，避免被调用的函数覆盖这些寄存器的值。
  2. 保存旧的`$fp`：将当前函数的帧指针（`$fp`）保存到栈上，为被调用函数准备新的栈帧环境。
  3. 保存返回地址`$ra`：`$ra` 寄存器存储了函数返回的地址，在调用另一个函数之前需要将其保存到栈上。
- **恢复操作**：

  函数执行完毕后，按照相反的顺序恢复`$ra`、`$fp` 和所有临时寄存器的值，保证调用者的环境不受影响。

图 3.6 是一个保存现场的示例：（假设调用函数时，只有 `$t0` 寄存器被占用。）

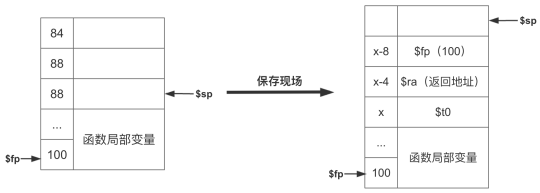

图 3.6 保存现场

然后执行 `move $fp, $sp`，此时由 `$fp` 和 `$sp` 管理的栈空间就是新函数的栈空间。

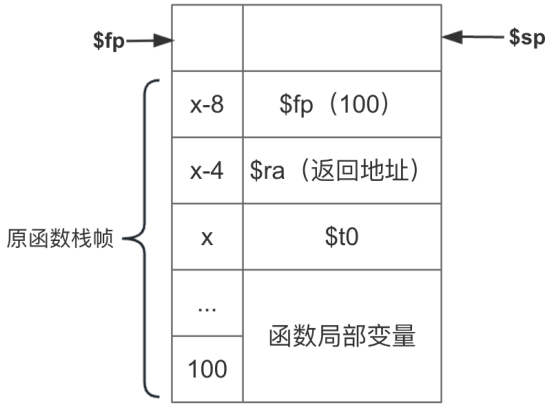

图 3.7 保存栈帧

B. 参数传递

MIPS 架构使用`$a0-$a3` 四个寄存器传递前四个参数，超出部分通过栈传递。

- 前四个参数：直接将参数值移动到`$a0-$a3` 中。
- 超过四个参数：在保存现场之后，将额外的参数依次压栈。在被调用函数中，可以通过基指针`$fp` 和偏移量访问这些参数。

函数在执行开始，将所有参数从参数寄存器`$a` 和栈空间中载入临时寄存器。

假设这个函数有 6 个参数，则调用后其栈空间和部分寄存器如下图：

C. 函数调用与返回

在每个函数开头增加一个 label，例如：

```
main:
    move $fp, $sp
    ...
    # 保存现场
    jal add
    # 恢复现场
    move $t0, $v0  # 获取返回值
    ...
    li $v0, 0 # return 0
    jr $ra    # 返回

add:
    move $fp, $sp
    ...
    move $v0, $t2 # return t2
    jr $ra
```

使用 `jal <函数名>`指令进行函数调用，`jal` 指令会自动将返回地址保存到`$ra` 寄存器。

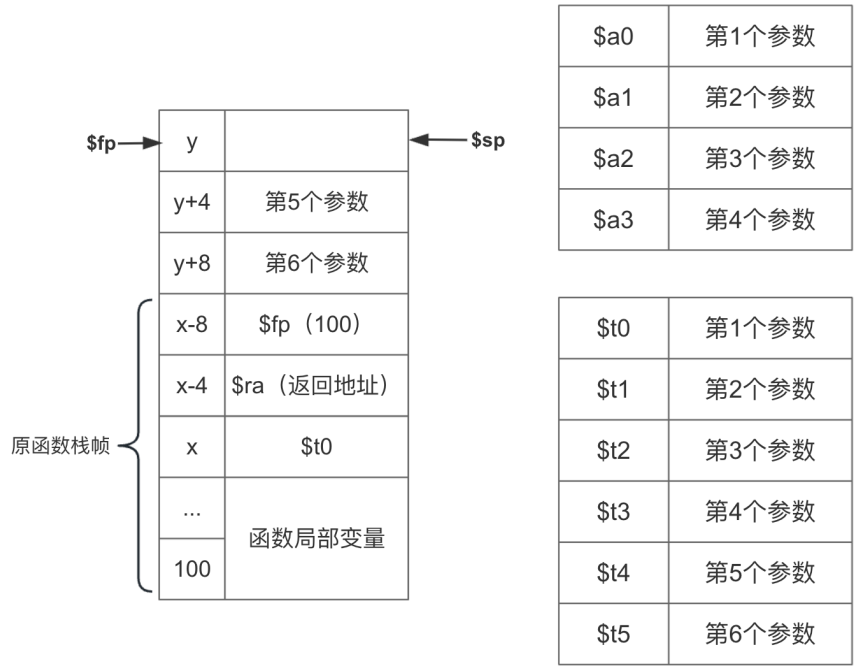

图 3.8 Enter Caption

`jal` 命令会保存当前的下一个指令地址到 `$ra` 寄存器，当函数返回时，调用 `jr $ra` 指令即可回到调用函数的下一条指令位置。

MIPS 预留了`$v0` 用作返回值传递。在返回之前，使用 `move $v0, $t0` 可以将返回值通过`$v0` 传递。

D. **增加预设文件头**

在编译生成的 MIPS 汇编代码中，通常会包含一个预设的文件头部分，这一部分定义了全局数据段、全局标签以及一些基础的输入输出函数。以下是该文件头的详细解析：

数据段

定义用于输入输出的字符串，例如提示输入的字符串`_prompt` 和用于换行的字符串`_ret`。

程序入口

- 使用`.globl main` 声明`main` 函数为全局入口点。
- 通过`jal main` 跳转到`main` 函数执行程序。
- 程序执行完`main` 函数后，使用`li $v0, 10` 和`syscall` 指令来正常退出程序。

输入输出函数

- `read` 函数用于读取用户输入的整数，首先输出提示信息，然后读取输入。
- `write` 函数用于输出整数和换行，将整数和换行符发送到标准输出。

这个文件头为 MIPS 汇编程序提供了执行前的初始化设置和常用的输入输出操作，确保程序的基本运行环境和功能。

```
        .data
                _prompt: .asciiz "Enter an integer:"
                _ret: .asciiz "\n"
        .globl main
        .text
                jal main
                li $v0, 10
                syscall
        read:
                li $v0, 4
                la $a0, _prompt
                syscall
                li $v0, 5
                syscall
                jr $ra
        write:
                li $v0, 1
                syscall
                li $v0, 4
                la $a0, _ret
                syscall
                move $v0, $0
                jr $ra
```

4 调试分析

为了验证编译器的正确性和编译结果的可靠性，使用 MARS（MIPS Assembler and Runtime Simulator）运行和调试生成的 MIPS 汇编代码。

MARS 是一个轻量级的 MIPS 汇编语言编程环境，它支持 MIPS 汇编代码的编辑、编译、运行和调试。

4.1 测试数据

- **测试 demo**（附件：input_demo.txt）
  运行结果与使用其他编译器（g++）的结果相匹配。

```
192

-- program is finished running --
```

图 4.1 测试 demo 运行结果

- **循环语句**（附件：input_累加.txt）
  运行结果与预期一致。
- **复杂表达式**（附件：input_复杂表达式.txt）
  运行结果与使用其他编译器（g++）的结果相匹配。
- **斐波那契数列**（附件：input_斐波那契数列.txt）
  用于测试函数递归。
  运行结果与预期一致。
- **阶乘**（附件：input_阶乘.txt）
  运行结果与预期一致。
  当输入过大时，会发生溢出。或与使用其他编译器（g++）的结果相匹配。
- **错误数据**（附件：input_错误.txt）
  定义变量 a 后，没有分号。
  输出结果：报语法错误“找不到语义动作”，并指出当前词法解析末尾位置，并输出最后两行。

装 订 线

```
19
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
-1

-- program is finished running --
```

图 4.2 循环语句运行结果

```
-121
-80
9698
-2256467

-- program is finished running --
```

图 4.3 复杂表达式运行结果

```
Enter an integer:5
5

-- program is finished running --


Reset: reset completed.

Enter an integer:10
55

-- program is finished running --


Reset: reset completed.

Enter an integer:15
610

-- program is finished running --
```

图 4.4 斐波那契数列运行结果

```
Enter an integer:5
120

-- program is finished running --


Reset: reset completed.

Enter an integer:10
3628800

-- program is finished running --
```

图 4.5 阶乘运行结果

```
Parse error: no action
编译失败
当前位置 row: 3 col: 7
2:      int a
3:      int b;
    -------^
```

图 4.6 错误数据运行结果

4.2 时间复杂度分析

4.2.1 词法分析

只需遍历文件中的所有字符，时间复杂度 $O(1)$

4.2.2 语法分析

构造 FIRST 集的时间复杂度

构造 FIRST 集包括初始化和迭代计算。对于每个非终结符，我们需要遍历其所有产生式，然后对每个产生式的右侧进行查找和合并操作。

- 假设有 N 个非终结符，P 个产生式，每个产生式的右侧平均长度为 R。
- 初始化的复杂度是 $O(N)$。
- 迭代计算的复杂度大致为 $O(P \times R)$，因为需要遍历所有产生式并查找其右侧符号的 FIRST 集。

构建 LR(1) 分析表的时间复杂度

构建分析表包括构建初始状态、状态转移、移进、归约、接受动作的处理。这个过程的复杂度取决于状态数、转移数、产生式数等因素。

- 假设最终构建的状态数为 S，每个状态平均有 T 个转移（包括移进和 GOTO 转移）。
- 构建初始状态和计算闭包的复杂度为 $O(P)$，因为需要考虑所有产生式。
- 对于每个状态，计算其转移和闭包的复杂度大约是 $O(T \times P)$，因为需要检查与现有状态的兼容性并可能添加新状态。
- 总的复杂度大致为 $O(S \times T \times P)$，这是因为需要为每个状态计算转移和闭包。

语法分析过程的时间复杂度

实际的语法分析过程依赖于输入的长度 L 和分析表的查找效率。

- 对于每个输入符号，需要执行一次查找操作，其复杂度是 $O(1)$，使用的数据结构是哈希表。
- 总的复杂度为 $O(L)$，因为每个输入符号最多处理一次。

因此，LR(1) 语法分析的整体时间复杂度主要受到构建分析表阶段的影响，这个阶段是最耗时的，尤其是当文法规模增大时。理论上，这个复杂度可能达到指数级，因为状态数 S 和转移数 T 可能随着文法的复杂度指数级增长。

4.2.3 语义分析

语义分析的时间复杂度主要取决于以下几个因素：

1. **符号表的管理**：包括查找、插入和删除符号表项的操作。

2. **属性的计算和传递**：在语法分析过程中，每个语法单元的属性值需要被计算和更新。

3. **中间代码的生成**：根据分析的结果生成中间代码，例如四元式。

符号表的管理

符号表通常采用哈希表来实现，以支持高效的查找、插入和删除操作，平均时间复杂度是 $O(1)$

属性的计算和传递

属性的计算和传递发生在语法分析的归约步骤中。假设平均每个归约动作需要 C 次属性计算。

对于输入长度为 L，理论上每个输入符号至少涉及一次归约，因此属性计算的总次数大约是 $O(C \times L)$。

中间代码的生成

中间代码的生成通常随着归约动作进行。每次归约可能会生成一个或多个四元式。

假设平均每次归约生成 M 个四元式，那么总的中间代码生成的复杂度也是 $O(M \times L)$。

综合分析

- 符号表操作的时间复杂度是 $O(L)$。
- 属性计算的时间复杂度是 $O(C \times L)$。
- 中间代码生成的时间复杂度是 $O(M \times L)$。

因此，语义分析的总体时间复杂度可以大致表示为 $O((C + M) \times L + L)$。对于大多数情况，可以认为是线性或接近线性复杂度，主要取决于输入长度 L，符号表的大小 N 以及每次归约所需要的属性计算次数 C 和生成的中间代码数量 M。

4.2.4 目标代码生成

目标代码生成阶段的时间复杂度分析涉及到几个关键步骤：寄存器分配、中间代码转换、栈空间管理、以及具体的指令生成过程。我们将分别考虑这些步骤的复杂度：

寄存器分配

- **寄存器分配策略**（线性扫描算法）的复杂度是对中间代码的一次遍历，即 $O(n)$ ，其中 n 是中间代码的数量。这个过程中，每个变量的生存期都会被考虑一次以决定其是否应该被分配到寄存器或溢出到栈。
- **寄存器溢出处理**：当活跃变量的数量超过可用寄存器时，某些变量需要被溢出到内存。这个决策过程也是线性的，因为每次溢出选择都是基于当前的活跃变量集。

中间代码转换

中间代码转换到目标代码的过程是逐条处理四元式，每条中间代码转换成一条或多条目标指令。因此，这个过程的复杂度也是 $O(n)$，其中 n 是中间代码的数量。

栈空间管理

栈空间的管理，包括在函数调用时保存现场、分配局部变量空间、以及恢复现场，主要是固定操作，其复杂度主要依赖于函数中局部变量和调用的数量，但对于每个函数，这个过程可以认为是常数时间，因此总的复杂度也是线性的，与函数调用的数量成正比。

指令生成

指令生成包括算术逻辑指令、跳转指令、以及函数调用指令等的生成。这个过程直接依赖于中间代码的数量和类型，每条中间代码转换为对应的一条或多条汇编指令。因此，指令生成的时间复杂度同样是 $O(n)$。

将上述各个阶段的复杂度加起来，目标代码生成阶段的总体时间复杂度也是线性的，即 $O(n)$，其中 n 是中间代码的数量。这个线性关系是因为每个阶段都涉及到对中间代码的一次或多次遍历，但没有任何阶段需要多于线性时间的操作（比如没有复杂的嵌套循环处理）。

4.3 问题分析与解决

4.3.1 语法分析过程

使用 Linux 中的 gprof 工具进行性能分析，发现程序最耗时的部分在于求闭包函数。求闭包函数耗时在于，频繁使用了 `std::set` 的插入操作，其 `insert` 方法包含去重算法，导致大量的无意义插入成为性能瓶颈。

A. **优化方案**：

为了解决这个问题，采取了以下优化措施：

- 引入一个布尔值 `changed` 来检查 `old_set` 是否有变化，以避免不必要的拷贝。
- 创建了一个 `items_to_add` 集合来存储本次循环中需要添加的元素，从而减少对 `old_set` 的直接修改。
- 在插入之前检查 `old_set` 是否已经包含了该元素，以减少不必要的插入操作。

优化后，插入操作在 `std::set` 中占用的时间显著减少，比较次数减少了 50%。

B. **可能的优化方案**：

进一步的优化方案包括：

- 使用 `std::unordered_set` 替换 `std::set`：基于哈希表实现的 `std::unordered_set` 在插入和查找操作上通常比基于红黑树的 `std::set` 更高效。
- 对于 FIRST 集的求解，最初采用递归方式可能导致左递归文法出现死循环。通过不断循环遍历每个产生式，并在一次遍历后如果 FIRST 集合没有变化则停止，避免了死循环问题。

4.3.2 目标代码生成

在循环结构中进行函数调用时出现的寄存器重新分配问题，主要是因为函数调用时为了保存现场而将临时寄存器中的变量溢出到内存，而在函数调用结束后恢复这些变量到寄存器时可能会因为

```cpp
void LR1Parser::closure(std::set<LR1Item> &old_set) const {
    bool changed = true;
    while (changed) {
        changed = false;
        std::set<LR1Item> items_to_add;
        for (const auto& item : old_set) {
            if (get_lr1item_state(item) != LR1Item::State::GOTO) continue;
            Symbol next_symbol = item.next_symbol();
            Symbol nnext_symbol = item.nnext_symbol();
            std::vector<Production> next_prods = get_productions_start_by_symbol(next_symbol);
            for (const auto& prod : next_prods) {
                std::set<Symbol> lookaheads = firstString({nnext_symbol, item.lookahead});
                for (const auto& p_lh : lookaheads) {
                    LR1Item new_item(prod, 0, p_lh);
                    if (old_set.find(new_item) == old_set.end()) {
                        items_to_add.insert(new_item);
                        changed = true;
                    }
                }
            }
        }
        old_set.insert(items_to_add.begin(), items_to_add.end());
    }
    return old_set;
}
```

寄存器的动态分配导致无法恢复到原先的寄存器。

解决方案：

1. **保存和恢复临时寄存器**：在函数调用前，保存所有使用中的临时寄存器状态，包括寄存器中存储的变量值。函数调用结束后，立即恢复这些寄存器的值。
2. **固定变量的栈空间分配**：为避免动态寄存器分配导致的问题，可以在函数起始部分为所有局部变量预先分配栈空间，并在函数执行过程中保持这一分配不变，确保即使变量在函数调用时被溢出到栈上也能保持它们地址的固定不变。

5 用户使用说明

本编译器提供了一个命令行界面，用户可以通过终端或命令提示符来运行和编译 C 类似语言的程序代码。以下是详细的使用方法说明：

5.1 命令行用法

```
C_Like_Compiler <输入文件> <输出位置> [-s 展示输出结果]
```

5.2 参数解析

- **<输入文件>**：指定要编译的源代码文件路径。这是用户编写的 C 类似语言程序文件。
- **<输出位置>**：指定编译后生成的汇编代码文件存储路径。编译器将处理结果保存在这个文件中。
- **[-s]**：这是一个可选参数。若指定了`-s` 参数，编译器除了将编译结果保存到指定的输出文件外，还会在控制台上展示中间代码和目标代码。若不使用`-s` 参数，则只会在控制台显示基本的编译信息，如编译成功或错误信息等。

5.3 示例使用

假设您有一个名为 `input.c` 的源代码文件位于`../input/`目录下，希望编译该文件并将生成的汇编代码保存到当前目录下的 `output.asm` 文件中，同时在控制台查看编译过程和结果，您可以这样操作：

```
C_Like_Compiler ../input/input.c ./output.asm -s
```

这条命令会启动编译器，对 `input.c` 进行编译，并将汇编代码输出到 `output.asm` 文件中。由于使用了`-s` 参数，中间代码和目标代码也会在控制台上显示。

参考文献

[1] OpenAI Chat. Available online: https://chat.openai.com.

[2] Zhihu. (2022). Flex(scanner)/Bison(parser) 词法语法分析工作原理. Available online: https://zhuanlan.zhihu.com/p/120812270.

[3] 编译原理-10039502. (2023). 第五章 语法分析——自下而上分析 [PowerPoint slides].

[4] AANA. (2022). 线性扫描寄存器分配（1）：基础介绍. Available online: https://www.cnblogs.com/AANA/p/16315921.html.

装 订 线